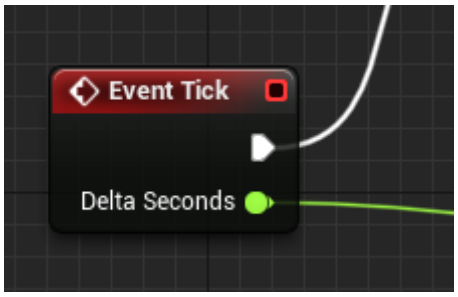
Steady Ticking

Almost all video-games (and thereby game engines) have a common technique for iterating on the game process. The game is basically run in a loop. This loop is thereby called a game loop. On each iteration of this loop the game should prepare and actualize the state of the game for rendering and perform the graphics rendering itself. These iterations are called Ticks within Unreal Engine. They are like a heartbeat to your game.

Like a human heartbeat, tick intervals can be varying in time, i.e. the time that it takes to process and render each frame is variable. You may suddenly have a computationally and/or graphically intensive frame, for example, and this in turn may result in a longer tick interval.

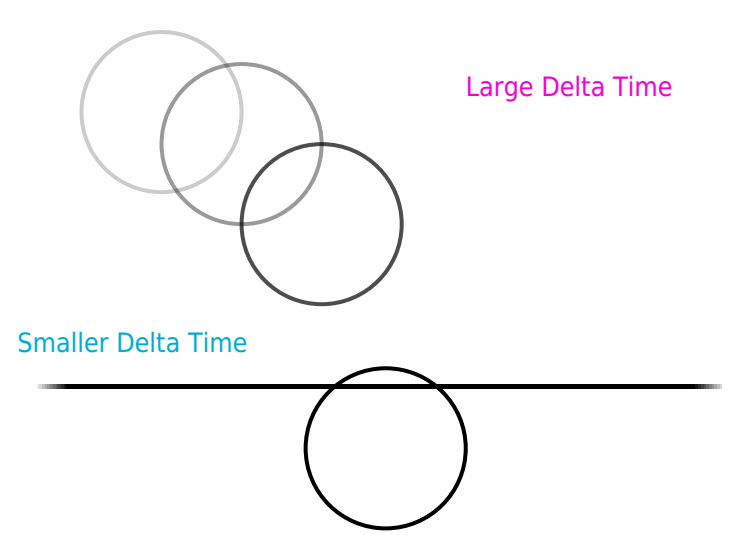
In order for the game logic to actualize the game state this variable delta time is usually passed to it to be able to “step” to this current frame from the previous one. This is implemented with a `Tick` event node and a `Delta Seconds` argument within Unreal Engine:



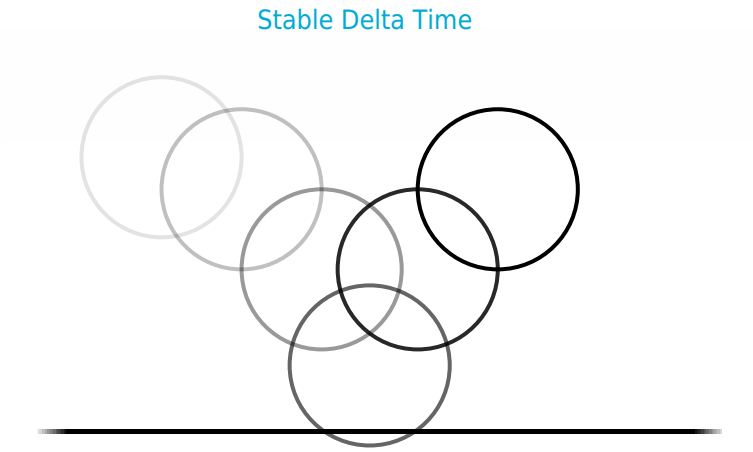
This technique is fine when you’re playing some kind of pre-baked animation where results are clearly determined in time. Unfortunately that is not always the case, since games are generally interactive and the situation always changes.

If we apply a variable frame rate to such concepts as collision detection and physics simulation, their stability will decrease considerably if the frame time is increased. Consider the following example.

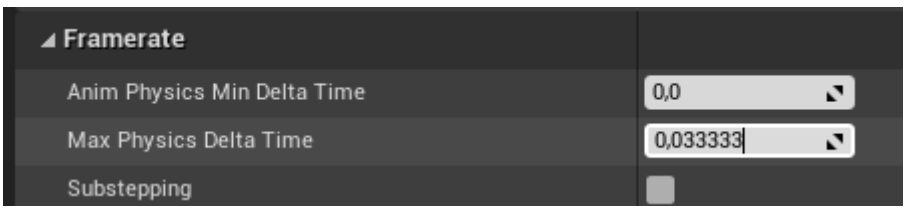
We have a ball physics simulation. The ball is stepping its position according to its velocity (which in turn results from gravity and other forces application). When the time step is stable and small enough, everything works as expected, but as soon as the one of the frames lags the system becomes unstable and the ball may actually fall through the ground, resulting in a totally inaccurate behavior.



Our goal would be to stabilize the frame time and in turn stabilize the calculations.



Unreal Engine's physics engine has a way to stabilize the frame rate by supplying the maximum frame time:

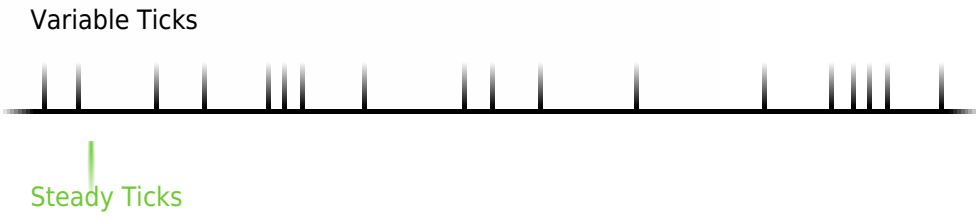


If the actual frame time is higher than this value, the physics engine will still receive this value as its delta time to step with. This in turn will result in a slowdown of game time. Something like a slow-mo effect, but the simulation would not lose its stability, i.e. it won't be lower than the supplied tolerated value.

Another way to achieve the physics stability effect is to use Unreal Engine's sub-stepping. You can read the linked official documentation for more information on that topic.

Apparatus provides a somewhat similar concept to a sub-stepping that is called “Steady Ticking”. Each of Mechanisms implements a special Steady Tick event:

Steady ticks are fixed-rate ticks that a run in parallel to the main variable-rate ticks. The “in parallel” wording isn’t related to multi-threading but merely a logical terminology.

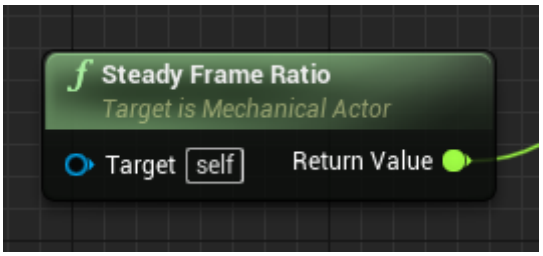


Each stable tick should actualize the

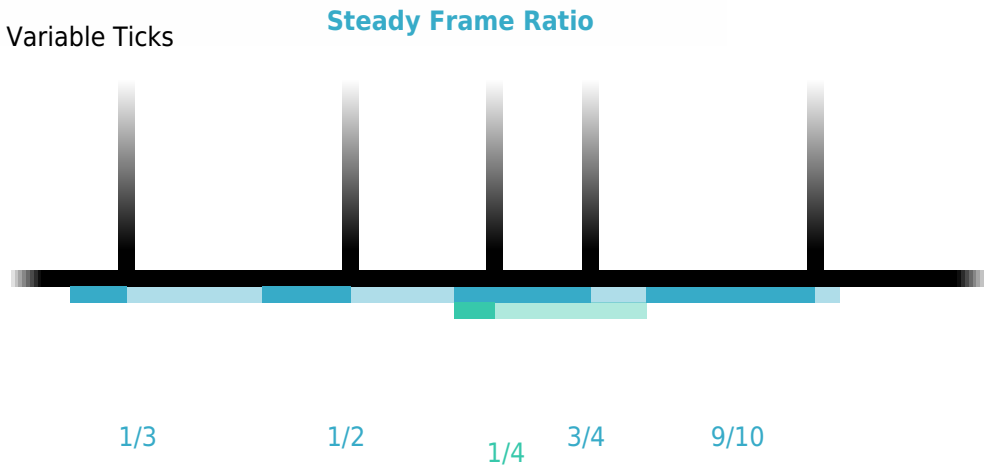
On the picture you can see that there may be multiple steady ticks during a single variable tick interval. Likewise there may be multiple variable ticks during a single steady tick interval. These multiple ticks can result in some undesired jerky movement. To solve this issue an interpolation is needed.

During the variable “native” ticking we have to interpolate between the previous steady state and the next one. It’s up to the user to do the actual interpolation but Apparatus provides a functional basis to do this.

First, there is a steady frame ratio node available:

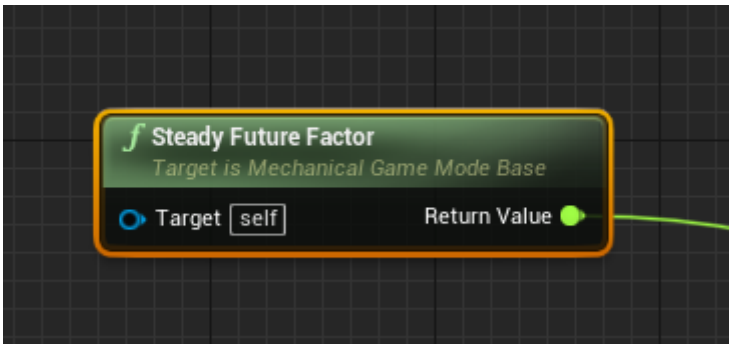


This node returns the ratio of the current variable tick in relation to an active steady frame it resides within. For a better understanding see the following schematic:

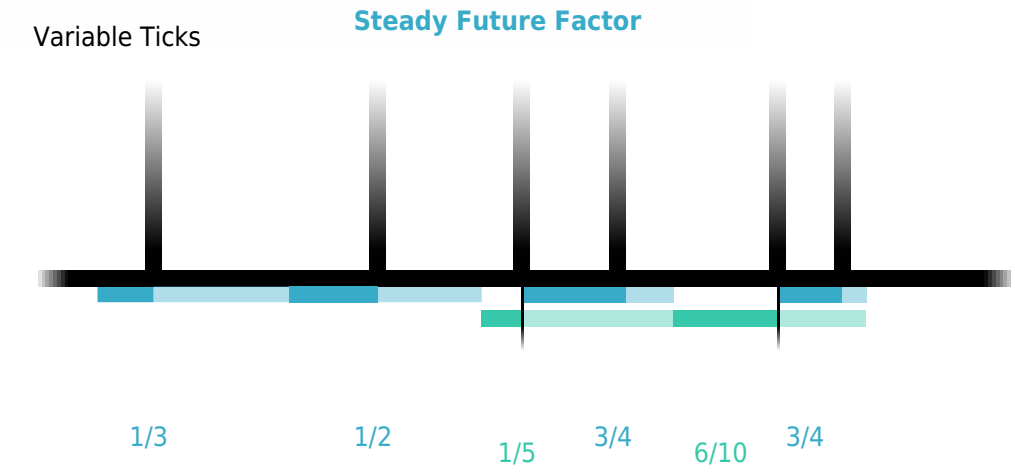


This ratio, which is bound from 0.0 to 1.0 (both inclusively), can be used to interpolate the animated visual feedback of some value based on its previous and the next state. In order to achieve that, your steady ticking mechanics have to prepare both states during their processing, while perhaps swapping the past state with the future one. The usual variadic ticking then can use a Lerp node to implement the actual smoothing.

Sometimes it's tedious (or kind of difficult) to manage both past and future states in some auxiliary separate variables. You may want to use the current object's state instead of its past one - store it in an Actor's current transform, for example. So we present an another way of smoothing things out - the Steady Future Factor node.

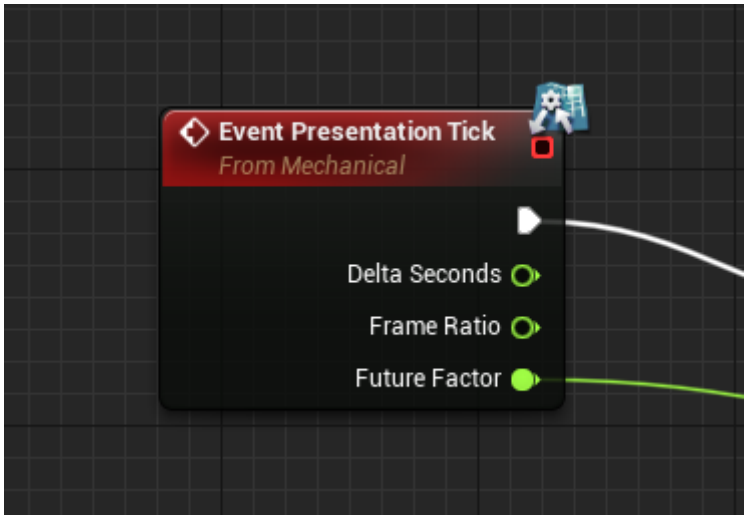


This node is similar to Steady Frame Ratio in that it is also returning a single float value bound to an inclusive [0.0, 1.0] range. It differs in the actual basis for this range. Let's illustrate it on the picture:



As you see, the last change (be it steady or a usual tick) is tracked and remembered as a starting point for the relation. The second part is the same as for the Steady Frame Factor node and actually represents a timestamp of the next future steady frame. Using the future factor you can use the current actual state as a base for Lerping with a next future state. Just be attentive to your calculations and make sure you actually keep track of what your are lerping to what.

Both of these nodes can be used directly during the Tick event in your mechanisms. However, we implemented a special type of mechanical event called Presentation Tick event, that essentially incorporates them both:



This event is run after your normal variable ticks (and hence after the necessary steady ticking) and is meant to be used for evaluating the visual feedback mechanics. Interpolating mechanics are logically and clearly one of those. Have a look at this complete position interpolation mechanic that is a part of our sample platformer project: